Wang and Liu *Cybersecur (2021)4:33*
https://doi.org/10.1186/s42400-021-00098-2

Cybersecurity

RESEARCH Open Access

# Position paper: GPT conjecture: understanding the trade-offs between granularity, performance and timeliness in control-flow integrity

Zhilong Wang* and Peng Liu

**Abstract**

Performance/security trade-off is widely noticed in CFI research, however, we observe that not every CFI scheme is subject to the trade-off. Motivated by the key observation, we ask three questions: ❶ does trade-off really exist in different CFI schemes? ❷ if trade-off do exist, how do previous works comply with it? ❸ how can it inspire future research? Although the three questions probably cannot be directly answered, they are inspiring. We find that a deeper understanding of the nature of the trade-off will help answer the three questions. Accordingly, we proposed the GPT conjecture to pinpoint the trade-off in designing CFI schemes, which says that at most two out of three properties (fine granularity, acceptable performance, and preventive protection) could be achieved.

**Keywords:** Conjecture, Control-flow integrity, Trade-off

## Introduction

Along with the increased complexity of software, it becomes harder for the developers to ensure execution correctness in their software products, especially in those developed by the low-level programming languages, such as C/C++. A substantial amount of execution incorrectness is caused by the exploitation of software vulnerabilities in the real world. Softwares inevitably contain a wide variety of vulnerabilities, opening a window for attacks to compromise the system. Attackers have developed a series of attack methods, such as shellcode injection (Erickson 2008), return-to-libc (Wojtczuk 2001), ROP (Shacham et al. 2007) and so on, to exploit all kinds of vulnerabilities, e.g., buffer overflow, format string, use-after-free, and so on (Szekeres et al. 2013). Among all kinds of attacks, the control-flow hijacking attack is the most dangerous one, because it allows the attacker to control the program's execution, execute arbitrary malicious code and attain Turing-complete operation (Shacham et al. 2007). To mitigate the threats, many defense mechanisms, such as stack smashing protector (SSP) (Cowan et al. 1998), address space layout randomization (ASLR) (Shacham et al. 2004), data execution prevention (DEP) (Andersen and Abella 2004) and so on, have been put forward by researchers and applied in the real world software products.

Among all the defense techniques, security schemes based on the concept of control-flow integrity (CFI) have attracted many researchers' attention because of its simplicity to implement, effectiveness to cope with the full spectrum of control-flow hijacking attacks, and flexibility to trade between security and efficiency. CFI schemes guarantee the correctness of the program by dynamically checking the control-flow transfer and confining the target address to a legal set.

Since CFI was introduced by Abadi et al. in 2005 (Abadi et al. 2005), many researchers afterward were dedicated to enhance its runtime performance, security, scalability, compatibility and so on. According to

*Correspondence: zzw169@psu.edu
The Pennsylvania State University, State College, USA

mainstream taxonomy, most CFI schemes can be clarified into two categories: fine-grained CFI schemes that provide more security guarantee, and coarse-grained CFI schemes that attain higher runtime performance. However, both fine-grained and coarse-grained CFI schemes have noticeable limitations that have not been addressed yet. As shown in previous survey papers (Burow et al. 2017), lightweight CFI schemes can not fully prevent sophisticated code reuse attack. The adversary's attacking strategy is to search large gadgets[1] chain whose starting addresses are allowed in a rough control-flow graph that coarse-grained CFI schemes adopted (Göktas et al. 2014; Lucas et al. 2014). Precise CFI schemes usually suffer from unacceptable runtime overhead. Hence, it is widely believed "performance/security trade-off" exists between runtime overhead and security in different CFI schemes (Burow et al. 2017; Xiaoyang et al. 2019).

However, we observe that not every CFI scheme is subject to the trade-off between performance and security. In fact, several CFI schemes are "immunized" from doing such a trade-off. For instance, $\pi$CFI designed by Niu et al. achieves fine-grained security with a runtime overhead of 3.2% on average, which is fairly low and acceptable (Niu and Tan 2015). Victor et al. proposed a context-sensitive CFI scheme that achieves stronger security than conventional fine-grained ones with an overhead of less than some of the coarse-grained ones (Victor et al. 2015).

*Key Observation.* The trade-off between performance and security does *not* universally exist in meaningful CFI schemes. This intriguing observation motivates us to ask three questions: ❶ does trade-off really exist in different CFI schemes? ❷ if trade-off do exist, how do previous works comply with it? ❸ how can it inspire future research?

Although the questions probably cannot be directly answered, they are inspiring. On the other hand, we find that a deeper understanding of the nature of the trade-off will help answer these questions. Accordingly, we propose the Gpt conjecture to pinpoint general trade-offs in CFI schemes: the impossibility of guaranteeing both fine granularity and acceptable performance in a Just-In-Time CFI scheme. We analyze its rationality through empirical study— surveying a series of representative CFI schemes and showing how existing CFI schemes comply with our conjecture. Finally, we give some recommendations for future researchers. We believe that our conjecture will help researchers have a more clear understanding of internal relations among properties of CFI schemes, thereby, motivating future research in this area.

## Background

When compiling source code written by low-level language (such as C or C++) into machine code, the compiler emits control data (Chen et al. 2005) (data that are loaded to processor program counter at some point in program execution, e.g., return addresses and function pointers) into the binary file without any protection. The security of control data depends on checks inserted by the programmer to enforce memory safety (Nagarakatte 2012). Along with program execution, attacker's malicious tampering with control data through software vulnerabilities, such as buffer overflow, can transfer the program's control-flow to any executable address in process space.

Based on this observation, researchers invented CFI to protect programs against control-flow hijacking attacks by checking programs' control data before loading them into the program counter (EIP/RIP register in x86/x64 architecture). CFI's strategy is to restrict the control-flow of a program to a pre-calculated CFG by checking indirect control-flow transfers at runtime (Burow et al. 2017). Generally, most of CFI schemes follow a mainstream that consists of two phases.

In ***phase one***, an analyzer statically computes the program's control-flow graph (CFG). CFG is a representation in graph form of all legitimate control-flow transfers (also being called branch) in program space. It consists of sets of nodes and directed edges. Each node and edge denotes a basic block and a valid branch in the program respectively. For a comprehensive understanding, we refer the reader to the formal definition of CFG in work by Allen (1970).

In ***phase two***, a runtime control-flow checking (validation) component validates just fetched control data before each indirect-branching according to the legitimate CFG generated in phase one.[2] An indirect-branch can pass checking only if it can be matched to a corresponding edge in the CFG. A failed validation will result in the process to terminate its execution and report an error. In such a fashion, control-flow attacks which usually introduce out-of-range branch are extremely prohibited. Researchers need to design efficient data structures to represent the CFG and enable runtime checking.

Despite its straightforward main idea, it is pretty challenging to design a CFI scheme with strong security, acceptable performance, high compatibility and so on (Xiaoyang et al. 2019; Burow et al. 2017). Researchers have designed hundreds of CFI schemes to explore its potential in different perspectives. The dominant

[1] Gadget is the terminology used in ROP attack (Shacham et al. 2007), which is the most important adversary of CFI.

[2] Direct control-flow transfers do not load any control data, their target addresses/offsets are hard-coded in their instructions.

difference of these various CFI schemes can be summarized into three aspects: (1) the precision of a CFG they employed. (2) the algorithm they designed to check indirect-branches. (3) the time point checking algorithm was activated.

#### Precision of CFG analyzer

CFG can be obtained by analyzing the program's source code or binary code. Like pointer analysis (Hind 2001), perfect CFG generation cannot be fully achieved yet in many situations (Göktas et al. 2014). By now researchers have adopted several types of methods (insensitive analysis, context-sensitive analysis, and path-sensitive analysis) in their CFG analyzer and achieve different precisions. It is widely agreed that path-sensitive analysis is more precise than context-sensitive analysis, and context-sensitive analysis is more precise than insensitive analysis (Khedker et al. 2017).

#### Algorithm to enforce checking

The efficiency of different CFI schemes is largely dependent on their algorithms to enforce validation, which is tightly combined with their data structure that represents the CFG and enables runtime checking. Researchers have designed different types of algorithms and data structures in different CFI schemes. For example, the original CFI scheme proposed by the Abadi, et al. groups branch targets into different sets, assigns each set with a label, and inlines labels into each jump targets, i.e., the basic block's in code. Based on this data structure, "guard instructions" are emitted before each indirect-branch instruction to compare its label with the one in target basic block (Abadi et al. 2005). A mismatch indicates that the control data is corrupted, then the program's execution will be redirect to the error handling code accordingly.

$\pi$CFI (Niu and Tan 2015) and MCFI (Niu and Tan 2014) by Niu, et al. adopts two ID tables, namely Bary and Tary, to store target program's CFG. In essence, Bary table and Tary table are hashmaps which can efficiently map indirect-branch points and target basic blocks to their corresponding IDs. Specifically, the Tary table is an array of IDs indexed by code addresses, mapping target basic block to their corresponding IDs. The Bary table uses a similar design, mapping indirect-branch points to their corresponding IDs. Two tables enable efficient ID look-ups and a indirect-branch is checked by comparing the IDs of branch point and target.

#### Just-in-time checking versus lazy checking

Another difference among CFI schemes is how they schedule their checking operations. Most CFI schemes check the target address before indirect-branch occurs (we define it as a `Just-In-Time checking`). While, to achieve better performance, some works log each indirect-branches at runtime and check them by employing another accompanying thread (Hu et al. 2018; Ding et al. 2017; Ge et al. 2017; Victor et al. 2015) (we define it as `Lazy checking`). For example, PITTYPAT (Ding et al. 2017) enforces path-sensitive CFI by maintaining a "shadow" execution/analyzer, running concurrently with the protected process and checking branches in execution trace of protected process. Such a non-intrusive checking does not disturb the normal execution of the monitored process, hence achieves path-sensitive CFI with practical runtime overhead.

## Conjecture

This section aims to answer Question❶ and Question❷. In this section, we firstly try to formalize the CFI enforcement problem. Secondly, we give a clear definitions of properties that are used to define our conjecture. Thirdly, we propose the GPT conjecture which helps to answer Question❶. Finally, some evidence is collected from an empirical study to answer the Question❷.

#### A definition of CFI enforcement problem

Before introducing the GPT conjecture, let us formalize CFI enforcement problem, that is how to enforce CFI during program running time. Our definition is try to define the problem itself. Therefore, the definition will not reflect any implementation of CFI schemes.

Given a program, let $\mathbb{S}$ denote the set of all indirect branches, and $\mathbb{T}$ denote the set of all targets of indirect branches. Then, $n = |\mathbb{S}|$ is the total number of indirect branches, and $m = |\mathbb{T}|$ is the total number of targets.

Let $\mathbb{S} \times \mathbb{T}$ denote the Cartesan Product of $\mathbb{S}$ and $\mathbb{T}$. Suppose that $s_i$ and $t_j$ are an element in $\mathbb{S}$ and $\mathbb{T}$, respectively. The CFI enforcement problem on a program is associated with a set $\mathbb{V}$, where $(s_i, t_j) \in \mathbb{V}$ if $t_j$ is an vaild target of branch $s_j$.

Based on the above notion, we can give a formal definition of CFI enforcement problem.

**Definition 1** CFI enforcement problem consists of two subproblems, which are corresponding to ***phase one*** and ***phase two*** discussed in "Background" section.

***Subproblem1:*** How to generate $\mathbb{V}$ through program analysis.

***Subproblem2:*** During program running time, how to verify whether a $(s_i, t_j)$, which corresponds to an encountered branch and target, is a member of set $\mathbb{V}$ or not.

In such a definition, the `granularity` is correlated with the size ($l$) of $\mathbb{V}$. The small the $l$ is, the finer

granularity a CFI scheme has. Each element (e.g., $(s_i, t_j)$) in set $\mathbb{V}$ represents a branch allowed by the CFI scheme. A smaller $\mathbb{V}$ means that fewer branches were allowed by the CFI scheme, and the CFI scheme has more tight constraints on the program's indirect branches, therefore a stronger security guarantee. The low bound of CFI `performance` is inherently connected with the complexity of subproblem2. Let $t_c$ and $t_b$ denote the check time and the branching time, respectively, for an indirect branch during problem execution. The `timeliness` is correlated with $t_c - t_b$. A CFI scheme is Just-In-Time if the $t_c - t_b < 0$, otherwise, it is an lazy-checking scheme.

#### Terminology

Even though the terms "fine-grained CFI" and "coarse-grained CFI" are well known and widely used by related researchers, we observe that they have not been clearly defined yet. In such a case, some so-called "fine-grained" schemes (claimed by their authors) are only relatively finer than some other schemes, but far from the finest-grained scheme. Therefore, in this subsection, we give clear definitions of such properties that are used to define our conjecture.

**Property 1** (Granularity) Suppose a program has $n$ indirect branch instructions. Let $\mathbb{Z}_i$ [3] denote the set of valid targets (successors) of the $i$-th indirect branch instruction, and $\mathcal{S}$ denote the set of all successor sets, namely,

$$\mathcal{S} = \{\mathbb{Z}_i : 1 \leq i \leq n\} \tag{1}$$

For a CFI scheme, let $\mathbb{C}_i$ denote the allowed target set which is defined by the scheme and assigned to the $i$-th indirect branch instruction, then used to check the branch's target at runtime. Only the elements in $\mathbb{C}_i$ are valid successors authorized by the CFI schemes that the $i$-th branch instruction could jump to.

**Definition 2** For arbitrary two sets $\mathbb{Z}_i$, $\mathbb{Z}_j$ from $\mathcal{S}$, satisfying $\mathbb{Z}_i \neq \mathbb{Z}_j$, as long as the CFI scheme merges any one or more $\mathbb{Z}_i$, $\mathbb{Z}_j$ when define its $\mathbb{C}_i$ or $\mathbb{C}_j$, namely,

$$\mathbb{C}_i = \mathbb{Z}_i \cup \mathbb{Z}_j \text{ or } \mathbb{C}_j = \mathbb{Z}_i \cup \mathbb{Z}_j \tag{2}$$

we define this scheme as a `coarse-grained CFI scheme`. Otherwise, we define it as a `fine-grained CFI scheme`. This definition enables us to determinate the `granularity` property of CFI schemes.

A coarse-grained CFI scheme will merge some $\mathbb{Z}$s when defining $\mathbb{C}$s (e.g., $\mathbb{C}_i = \mathbb{C}_j = \mathbb{Z}_i \cup \mathbb{Z}_j$). In such a case, the set $\mathbb{C}_i$ used to enforce runtime checking is a superset of the valid target set $\mathbb{Z}_i$. A potential attacker can hijack program's runtime control flow to a gadget—$x$ ($x \in \mathbb{C}_i \wedge x \notin \mathbb{C}_j$), and without being detection by such a CFI scheme. "Otherwise" means the CFI scheme does not merges any $\mathbb{Z}$s when defining $\mathbb{C}$s. Therefore, any $\mathbb{C}_i$ is strictly equal to $\mathbb{Z}_i$. In such a case, an attacker cannot find any gadget—$x$, which belongs to $\mathbb{C}_i$ but not belongs to $\mathbb{Z}_i$.

*Remark 1.* According to Definition 2, both context-sensitive and path-sensitive CFI schemes belong to `fine-grained CFI scheme`. In essence, they reduce the size of their checking set $\mathbb{C}_i$ for $i \in [1, n]$ based on context-sensitive or path-sensitive pointer analysis. Their protection is generally considered to be more powerful than that of insensitive `fine-grained CFI scheme`.

*Remark 2.* Note that CFI schemes (Mashtizadeh et al. 2015; Zhang et al. 2019) which adopt pointer encryption approach should be classified as `coarse-grained CFI scheme`. They cannot fully prevent code reuse attack because of two noticeable drawbacks. As discussed in Cryptographically Enforced Control Flow Integrity (CCFI) (Mashtizadeh et al. 2015), it is still possible to replace the current encrypted pointer with another one from the program space and potentially disrupt control flow. The other drawback is that these schemes suffer from key leakage issues: the key can be infered by brute-force attack or known-plaintext attack (Peng et al. 2006), especially for schemes which adopt a very simple encryption/decryption method (e.g., XOR) (Zhang et al. 2019).

*Remark 3.* We remark that schemes that only provide partial protection—protecting subset of indirect branches in program space—belong to `coarse-grained CFI scheme`. For instance, `vfGuard` (Prakash et al. 2015), VTV (Tice et al. 2014), and SafeDispatch (Jang et al. 2014) only achieve strict protection for virtual function calls in COTS binaries;

**Property 2** (Performance)

*Evidence 1.* As discussed in many papers (Szekeres et al. 2013; Burow et al. 2017; Starr and Abella 2012), runtime performance is one of the most important determinants of whether a defense technique will be adopted by industry. Generally, to get adopted by industry, a defense technique should introduce less than 5% average overhead, such as StackGurad, ASLR, and DEP. Techniques incuring an overhead larger than 10% do not tend to gain wide adoption in production environments. Accordingly, the threshold should lie between 5%-10%.

[3] It is computed through mainstream insensitive control flow analysis. We admit the inaccuracy due to the difficulty of the pointer analysis.

*Evidence 2.* Other than runtime performance, space performance is another important index to measure a scheme. Program's runtime memory consumption consists of four aspects, i.e., code, global data, heap, and stack. Different programs have different ratios in four aspects, and a defense technique commonly increases memory consumption in one or more aspects. We observe that shadow based protections like shadow stack (Dang et al. 2015), shadow memory (Newsome and Song 2005) and shadow processing (Patil and Fischer 1995), that double memory consumption in one or more aspects are unlikely to be deployed in practice.

**Definition 3** Conservatively, we define a runtime overhead of less than 10% and a space overhead of less than 100% (in any of aforementioned four aspects) as an `acceptable performance`. Otherwise, it is an `unacceptable performance`. This definition enables us to determinate the `performance` property of CFI schemes.

**Property 3** (Timeliness)
*Observation 1.* Whereas the term "integrity" in the context of CFI implies that it can prevent the attacks (Abadi et al. 2005), some of the CFI schemes do not hit the mark. To achieve higher efficiency, some CFI schemes as mentioned in "Just-in-time checking versus lazy checking" section adopted a `lazy checking` mechanism, which checks programs' control-flow following the program's execution rather than before each indirect branching. Generally, they log the program's runtime control-flow transfer along with its execution, then check the control-flow offline or through an accompanying thread. In these designs, a sliding window exists between the program's control-flow transfer and checking. The attacker can compromise the system without being perceived in the sliding window, which means this kind of CFI cannot protect software against such attacks.

**Definition 4** We regard that the aforementioned design of CFI schemes provides less protection than CFI schemes that perform `Just-In-Time checking`. We define protection capability powered by `lazy checking` schemes as `detective protection`, the others that powered by `Just-In-Time checking` as `preventive protection`. This definition enables us to determinate the property `timeliness` of CFI schemes.

### An empirical analysis of the relation among three properties

We notice a strong relation between granularity, performance, and timeliness. In this subsection, we will provide some empirical observations regarding the relationship among three properties.

Firstly, the trade-off between granularity and performance (time/space overhead) has been widely noticed by previous researchers (Tice et al. 2014; Burow et al. 2017). A finer granularity means less $\mathbb{Z}$s was merged when defining $\mathbb{C}$s. It indicates that more equivalent sets $\mathbb{C}$s were used to enforce CFI. Generally, more equivalent sets will require more time/space when designing an algorithm to enforce checking.

Secondly, trade-off between time and space commonly exists when designing algorithms to solve a problem. For the membership problem, the most effective way to solve it is through a hash table, which maps all possible queries to the corresponding results directly. In such a case, the runtime overhead is negligible (O(1)), however, the space overhead is fairly large. Another algorithm, which answers queries through traversing lists, will require less memory, but suffer from a larger time overhead.

Thirdly, we observe the trade-off between timeliness and performance. Lazy checking CFI schemes validate the target address after indirect-branch occurs. We observed that the main purpose of such a design is to reduce runtime overhead. Because the task of runtime checking in such a case could be assigned to another monitor thread, thus greatly reduce the runtime overhead of the protected program. If we measure the timeliness through $t_c - t_b$. Then, if $t_c - t_b < 0$, the CFI scheme achieves just-in-time checking. Otherwise, it achieves lazy checking. Meanwhile, a less timeliness (the larger $t_c - t_b$ is), the fewer synchronizations are required between the program's thread and the monitor thread. Fewer synchronizations mean a small overhead to the protected program. Therefore, there exists a trade-off between timeliness and performance

Finally, both the granularity and timeliness will affect the security of a CFI scheme in different aspects. Specifically, attackers need both a certain amount of gadgets and a certain amount of time for code reuse attacks (e.g., ROP (Shacham et al. 2007)). A finer granularity better restricts the number of gadgets that attackers can use. Better timeliness (the small $t_c - t_b$ is) better restricts the time that attackers can use. Therefore, both finer granularity and better timeliness mean fewer risks that the CFI protection can be compromised.

### The proposed conjecture

> **`GPT Conjecture:`** A control flow integrity scheme, which is developed to solve the CFI enforcement problem defined in Definition 1, can have at most two out of three properties:
> **`P1.`** `Fine granularity` defined in Definition 2
> **`P2.`** `Acceptable performance` defined in Definition 3
> **`P3.`** `Preventive protection` defined in Definition 4

### Some evidence of the GPT conjecture

In this section, we will reflect on our conjecture through several pieces of evidence. To verify the rationality of our conjecture, we conduct an empirical study on 32 representative works, and show the results in Table 1. Three columns (**`P1`**, **`P2`** and **`P3`**) in the table display three properties respectively as we define in "Terminology" section. **`P1`** column denotes the granularity— check-mark indicates a `fine-grained scheme` whileas cross-mark represents a `coarse-grained scheme`. **`P2`** column shows the performance overheads which are reported in corresponding papers. Note that we prefer evaluation results which are based on SPEC CPU®2006 benchmarks (SPEC 2006). **`P3`** column labels whether a CFI scheme provides `preventive protection`. We label the data in each column with *red* color when it fails to meet the requirement defined in the conjecture.

*Evidence i.* It can be clearly seen in Table 1 that all CFI schemes we surveyed comply with our conjecture—**no CFI schemes can achieve all three properties**. Also, some of unsophisticated schemes, such as PittyPat (Ding et al. 2017) and Griffin (Ge et al. 2017), only achieve one properity, i.e., `fine granularity`.

*Evidence ii.* MCFI (Niu and Tan 2014) and $\pi$CFI developed by Niu, et al. achieve `fine granularity` with acceptable runtime overheads, i.e., 3.2% and 5.0%, respectively. However, researchers did not realize that their better runtime overhead is achieved through sacrificing their space performance. Even though they did not report their space overhead in their paper explicitly, we can infer it in a reasonable manner.

As discussed in "Algorithm to enforce checking" section, both of two schemes adopt two tables, namely Bary and Tary, to support their runtime checking. Accordingly, 1 GB/4 GB memory space on x86-32 and x86-64 operating system, respectively, need to be reserved in each process for the tables. As stated by the author, "On x86-32, memory segmentation is used, as in NaCl (Yee et al. 2009). A 1 GB segment is reserved for running the application code and another 1 GB segment is reserved for the table region. x86-64, however, does not support memory segmentation. Instead, memory writes are instrumented so that they are restricted to the [0, 4 GB) memory region. Another 4 GB memory region is reserved for tables." In view of the size of memory consumption of typical programs (mostly less than 1 GB (SPEC 2006)),

**Table 1** Reflection of GPT conjecture in 32 control-flow integrity schemes

| Schemes | | P1[1] | P2 | P3 |
|---|---|---|---|---|
| CFIXX [34] | | ✗ | 4.98% | ✓ |
| Reins [35] | | ✗ | 2.40% | ✓ |
| vfGuard [27] | | ✗ | 18.30% | ✓ |
| VTV [28] | | ✗ | 9.60% | ✓ |
| LLVM-CFI [36] | | ✗ | 1.10% | ✓ |
| VTI [37] | | ✗ | 0.50% | ✓ |
| CFGuard [38] | | ✗ | 2.30% | ✓ |
| IFCC [28] | | ✗ | -0.30% | ✓ |
| ROPecker [39] | | ✗ | 2.60% | ✓ |
| bin-CFI [40] | | ✗ | 8.50% | ✓ |
| ROPGuard [41] | | ✗ | 0.48% | ✓ |
| SafeDispatch [29] | | ✗ | 2.00% | ✓ |
| CCFIR [42] | | ✗ | 2.08% | ✓ |
| kBouncer [43] | | ✗ | 4.00% | ✓ |
| OCFI [44] | | ✗ | 4.70% | ✓ |
| CFIMon [45] | | ✗ | 6.10% | ✓ |
| $\tau$CFI [46] | | ✗ | 2.89% | ✓ |
| HCIC [25] | | ✗ | 0.95% | ✓ |
| RAGuard [47] | | ✗ | 1.86% | ✓ |
| HyperSafe [48] | | ✗ | 5.00% | ✓ |
| BinCC [48] | | ✗ | 4.00% | ✓ |
| CCFI [24] | | ✗ | 52.00% | ✓ |
| KCoFI [49] | | ✗ | 13.00% | ✓ |
| Original CFI [50] | | ✓ | 16.00% | ✓ |
| Lockdown [51] | | ✓ | 20.00% | ✓ |
| MCFI [20] | | ✗ | 5.00% & 4GB | ✓ |
| $\pi$CFI [13] | | ✗ | 3.20% & 4GB | ✓ |
| Griffin [23] | H[2] | ✓ | 11.90% | ✗ |
| PittyPat [22] | P, H | ✓ | 12.73% | ✗ |
| ECFI [52] | | ✓ | 1.50% | ✗ |
| $\mu$CFI [21] | C, H | ✓ | 10.00% | ✗ |
| PathArmor [14] | C | ✓ | 3.00% | ✗ |

[1] If a CFI scheme supports different security levels, e.g. having both `coarse-grained` and `fine-grained` versions, we focus on its most secure version

[2] 'H', 'P' and 'C' denote hardware-assisted CFI scheme, path sensitive CFI scheme, and context sensitive CFI scheme, respectively

their space overhead has already reached 100% except for code bloat caused by extra no-op instructions inserted to enforce four-byte alignment on indirect-branch targets.

*Evidence iii.* GRIFFIN (Ge et al. 2017) is a hardware-assisted CFI, which leverages Intel PT to record control-flow of a monitored program. It supports multiple types of CFI policies to enable flexible trade-offs between security and performance. The `fine-grained` scheme incures an average of 11.9% overhead. It leverages idle cores on a multi-core system for security checking by having multiple worker threads to check runtime control-flow simultaneously. In most of the time, it performs non-blocking checking which analyzes trace buffer of Intel PT whenever it becomes full; In a few cases when security-sensitive system calls are invoked, it performs blocking checking which stops the target thread until all the control transfers in the buffer have been checked. It can only provide the `detective protection` for software according to Definition 4. This case indicates that GPT conjecture is applicable to hardware-assisted CFI schemes.

*Evidence iv.* PITTYPAT (Ding et al. 2017), $\mu$CFI (Hu et al. 2018) and PathArmor (Victor et al. 2015) are path/context sensitive CFI schemes which adopt path-sensitive or context-sensitive analysis to generate their CFG. However, path-sensitive and context-sensitive analysis is generally considered to be more time-consuming and space-consuming than insensitive analysis (Khedker et al. 2017). We find that all three CFI schemes adopt two common features: `hard-assisted branch recording` and `lazy checking`. Specifically, PITTYPAT and $\mu$CFI employ Intel PT—a brand new hardware feature in Intel CPUs—to efficiently record conditional and indirect branches taken by a program at runtime while PathArmor adopts Last Branch Record (LBR) registers available in Intel processors to monitor recently exercised control-flow transfers in an efficient way. Their control-flow checking is achieved through accompanying threads. This case indicates that both path-sensitive and context-sensitive CFI schemes conform to the claim of GPT conjecture.

*Remark 4.* Our observations indicate that the GPT conjecture is universally applicable in all kinds of scenarios. Further, four pieces of evidence are not meant to be exhaustive and more evidence are easy to find.

## Implications of the GPT conjecture

In this section, we will focus on answering Question ❸: how can GPT conjecture inspire future research?

First of all, GPT conjecture illustrates the inherent trade-offs of three important properties (`fine granularity`, `acceptable performance`, and `preventive protection`) in CFI schemes. It helps researchers to have a deeper understanding of the nature of CFI based protection. Accordingly, future researchers should make a necessary sacrifice before designing new CFI schemes. In the broader context, GPT conjecture provides insights into the feasible design space for CFI schemes, shedding some light on the manner in which algorithm designers and software engineers have circumvented the conjecture.

Second, for decades, security researchers have been focused on CFI scheme's runtime performance and made their best effort to improve it. Evidence ii shows that in some cases, better runtime performance is achieved by sacrificing its space performance. Just as Gerhard states, "For some problems, we can reach an improved time complexity, but it seems that we have to pay for this with an exponential space complexity" (Woeginger 2004). Therefore, performance evaluation in future research should not merely be limited to runtime performance and researchers should have a more comprehensive evaluation of their schemes.

Third, Evidence iii shows that even powerful hardware support cannot eliminate the runtime overhead of `Just-In-Time` CFI schemes to an acceptable level, which implies that the challenge in the implementation of CFI cannot be solved only through engineering efforts, instead, it may relate to computational complexity theory (Goldreich 2008). In a broader sense, we observe that indirect branching poses not only challenge in the security field, but also challenges to many others: precise pointer analysis is NP-hard (Horwitz 1997); indirect branch prediction is a performance-limiting factor for current computer systems (Santana et al. 2002). Hence, GPT conjecture implies the complexity of the CFI enforcement problem, which deserves to be investigated through theoretical methods.

Fourth, the GPT conjecture could have the following impact in practice: firstly, for CFI product managers, engineers, the conjecture provides useful awareness. That is the research works have not yet achieved P1, P2, and P3 simultaneously. Secondly, the conjecture provides awareness on 3 trade-offs in developing CFI products. For example, the LLVM CFI choose to implement different options for user to trade-off between performance and granularity.

At last, despite the inspiring implications that GPT conjecture gives to us, we admit that we still cannot prove the conjecture at this time.

### Some suggestions for future research

In most cases, *Subproblem1* in definition 1 can be solved through offline static analysis. The performance and timeliness of a CFI scheme depend on the algorithm to solving *Subproblem2*. Our definition of CFI enforcement problem reveals that the *Subproblem2* (online phase) can be viewed as *membership problem* (Mastrolilli 2021). However, even though the membership problem has

been widely researched in theoretical computer science, it does not mean that the conjecture can be easily proved.

Also, we think that the proof of GPT conjecture need some statistical data from benchmark programs. For example, the average size of $\mathbb{S}$ and $\mathbb{T}$ in benchmark programs, the variance of $s_i$ and $t_j$, and the frequency of indirect jumps to be executed.

## Conclusion

Control-flow integrity is a popular defence technique for detecting and defeating control-flow hijacking attacks. Since its inception in the decade, researchers have put great efforts to explore its potential regarding security, performance, compatibility and so on. Even though performance/security trade-off is widely noticed in CFI research, we observe that not every CFI scheme is subject to it. In this paper, we propose the GPT conjecture to illustrate the general trade-offs in CFI schemes. The conjecture points out the impossibility of guaranteeing both `fine granularity` and `acceptable performance` in a `Just-In-Time` CFI schemes. We have verified the rationality of our conjecture based on an empirical study on existing works. Even though we cannot prove the conjecture at this time, we believe that GPT conjecture will help researcher to have a deeper understanding of the nature of CFI enforcement problem and it will direct future research in this area.

**Acknowledgements**
Not applicable.

**Authors' contributions**
All authors read and approved the final manuscript.

**Funding**
This work was supported by ARO W911NF-13-1-0421 (MURI), NSF CNS-1814679, and NSF CNS-2019340.

**Availability of data and materials**
Not applicable.

## Declarations

**Competing interests**
The authors declare that they have no competing interests.

Received: 14 May 2021 Accepted: 9 August 2021
Published online: 16 September 2021

**References**

Abadi M, Budiu M, Erlingsson Ú, Ligatti J (2009) Control-flow integrity principles, implementations, and applications. ACM Trans Inf Syst Secur (TISSEC) 13(1):4

Abadi M, Budiu M, Erlingsson Ú, Ligatti J (2005) Control-flow integrity. In: Proceedings of the 12th ACM conference on computer and communications security, CCS '05, New York, NY, USA, ACM, pp 340–353

Abbasi A, Holz T, Zambon E, Etalle S (2017) ECFI: asynchronous control flow integrity for programmable logic controllers. In: Proceedings of the 33rd annual computer security applications conference, ACSAC 2017, New York, NY, USA, ACM, pp 437–448

Allen FE (1970) Control flow analysis. In: Proceedings of a symposium on compiler optimization, New York, NY, USA, ACM, pp 1–19

Andersen S, Abella V (2004) Data execution prevention. Changes to functionality in Microsoft Windows XP Service Pack 2, Part 3. Memory protection technologies

Bounov D, Kici RGö, Lerner S (2016) Protecting C++ dynamic dispatch through VTable interleaving. In: The network and distributed system security symposium (NDSS)

Burow N, Carr SA, Nash J, Larsen P, Franz M, Brunthaler S, Payer M (2017) Control-flow integrity: precision, security, and performance. ACM Comput Surv 50(1):16

Burow N, McKee D, Carr SA, Payer M (2018) CfIXX: object type integrity for C++ virtual dispatch. In: Proceedings of network and distributed system security symposium (NDSS). https://hexhive.epfl.ch/publications/files/18NDSS.pdf

Cheng Y, Zhou Z, Miao Y, Ding X, Deng RH (2014) ROPecker: a generic and practical approach for defending against ROP attack. In: Symposium on network and distributed system security (NDSS). Internet Society

Chen S, Xu J, Sezer EC, Gauriar P, Iyer RK (2005) Non-control-data attacks are realistic threats. In: USENIX security symposium, vol 5

Cowan C, Calton P, Maier D, Walpole J, Bakke P, Beattie S, Grier A, Wagle P, Zhang Q, Hinton H (1998) Stackguard: automatic adaptive detection oand prevention of buffer-overflow attacks. In: USENIX security symposium. San Antonio, TX

Criswell J, Dautenhahn N, Adve V (2014) KCoFI: complete control-flow integrity for commodity operating system kernels. In: 2014 IEEE symposium on security and privacy, pp 292–307

Dang THY, Maniatis P, Wagner D (2015) The performance cost of shadow stacks and stack canaries. In: Proceedings of the 10th ACM symposium on information, computer and communications security (ASIACCS 15), New York, NY, USA, ACM, pp 555–566

Ding R, Qian C, Song C, Harris B, Kim T, Lee W (2017) Efficient protection of path-sensitive control security. In: 26th USENIX security symposium (USENIX security 17), Vancouver, BC, USENIX Association, pp 131–148

Erickson J (2008) Hacking: the art of exploitation

Fratrić I (2012) ROPGuard: runtime prevention of return-oriented programming attacks. Technical report

Ge X, Cui W, Jaeger T (2017) GRIFFIN: guarding control flows using intel processor trace. In: Proceedings of the twenty-second international conference on architectural support for programming languages and operating systems (ASPLOS 17), New York, NY, USA, ACM, pp 585–598

Göktas E, Athanasopoulos E, Bos H, Portokalidis G (2014) Out of control: overcoming control-flow integrity. In: 2014 IEEE symposium on security and privacy (S&P). IEEE

Goldreich O (2008) Computational complexity: a conceptual perspective. SIGACT News 39(3):35–39

Hind M (2001) Pointer analysis: Haven'T we solved this problem yet? In: Proceedings of the 2001 ACM SIGPLAN-SIGSOFT workshop on program analysis for software tools and engineering, PASTE '01, New York, NY, USA, ACM, pp 54–61

Horwitz S (1997) Precise flow-insensitive may-alias analysis is NP-hard. ACM Trans Program Lang Syst 19(1):1–6

Hu H, Qian C, Yagemann C, Chung SPH, Harris WR, Kim T, Lee W (2018) Enforcing unique code target property for control-flow integrity. In: Proceedings of the 2018 ACM SIGSAC conference on computer and communications security. ACM, pp 1470–1486

Jang D, Tatlock Z, Lerner S (2014) SafeDispatch: securing C++ virtual calls from memory corruption attacks. In: the Network and distributed system security symposium (NDSS)

Khedker U, Sanyal A, Sathe B (2017) Theory and practice, data flow analysis

LLVM—control flow integrity (2015)

Lucas D, Sadeghi A-R, Lehmann D, Monrose F (2014) Stitching the gadgets: on the ineffectiveness of coarse-grained control-flow integrity protection. In: 23rd USENIX security symposium (USENIX Security 14). USENIX

Mashtizadeh AJ, Bittau A, Boneh D, Mazières D (2015) CCFI: cryptographically enforced control flow integrity. In: Proceedings of the 22nd ACM SIGSAC conference on computer and communications security. ACM, pp 941–951

Mastrolilli M (2021) The complexity of the ideal membership problem for constrained problems over the Boolean domain
Microsoft. Visual Studio 2015—compiler options—enable control flow guard (2015)
Mohan V, Larsen P, Brunthaler S, Hamlen KW, Franz M (2015) Opaque control-flow integrity. In: The network and distributed system security symposium (NDSS), vol 26, pp 27–30
Muntean P, Fischer M, Tan G, Lin Z, Grossklags J, Eckert C (2018) $\tau$ CFI: type-assisted control flow integrity for x86-64 binaries. In: International symposium on research in attacks, intrusions, and defenses. Springer, pp 423–444
Nagarakatte SG (2012) Practical low-overhead enforcement of memory safety for C programs
Newsome J, Song DX (2005) Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In: The network and distributed system security symposium (NDSS). Citeseer, vol 5, pp 3–4
Niu B, Tan G (2014) Modular control-flow integrity. In: Proceedings of the 35th ACM SIGPLAN conference on programming language design and implementation (PLDI 14), New York, NY, USA. ACM, pp 577–587
Niu B, Tan G (2015) Per-input control-flow integrity. In: Proceedings of the 22nd ACM SIGSAC conference on computer and communications security, CCS '15, New York, NY, USA, ACM, pp 914–926
Pappas V, Polychronakis M, Keromytis AD (2013) Transparent ROP exploit mitigation using indirect branch tracing. In: Proceeding of the 22nd USENIX security symposium (USENIX security 13), pp 447–462
Patil H, Fischer CN (1995) Efficient run-time monitoring using shadow processing. In: Proceeding of automated and algorithmic debugging (AADEBUG), vol 95, pp 1–14
Payer M, Barresi A, Gross TR (2015) Fine-grained control-flow integrity through binary hardening. In: International conference on detection of intrusions and malware, and vulnerability assessment. Springer, pp 144–164
Peng X, Zhang P, Wei H, Bin Yu (2006) Known-plaintext attack on optical encryption based on double random phase keys. Opt Lett 31(8):1044–1046
Prakash A, Hu X, Yin H (2015) vfGuard: strict protection for virtual function calls in COTS C++ binaries. In: Symposium on network and distributed system security (NDSS)
Santana OJ, Falcón A, Fernández E, Medina P, Ramírez A, Valero M (2002) A comprehensive analysis of indirect branch prediction. In: Hans PZ, Kazuki J, Mitsuhisa S, Yoshiki S, Masaaki S (eds) High performance computing. Springer, Berlin, pp 133–145
Shacham H et al (2007) The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86). In: ACM conference on computer and communications security. New York, pp 552–561
Shacham H, Matthew P, Ben P, Eu-Jin G, Nagendra M, Dan B (2004) On the effectiveness of address-space randomization. In: Proceedings of the 11th ACM conference on computer and communications security. ACM
SPEC CPU 2006 system requirements. https://www.spec.org/cpu2006
Starr A, Abella V (2012) The BlueHat prize contest official rules
Szekeres L, Payer M, Wei T, Song DS (2013) Sok: eternal war in memory. In: 2013 IEEE symposium on security and privacy (S&P). IEEE, pp 48–62
Tice C, Roeder T, Collingbourne P, Checkoway S, Erlingsson Ú, Lozano L, Pike G (2014) Enforcing forward-edge control-flow integrity in GCC & LLVM. In: 23rd USENIX security symposium (USENIX security 14), pp 941–955
Victor Van der V, Andriesse D, Göktaş E, Gras B, Sambuc L, Slowinska A, Bos H, Giuffrida C (2015) Practical context-sensitive CFI. In: Proceedings of the 22nd ACM SIGSAC conference on computer and communications security. ACM, pp 927–940
Wartell R, Mohan V, Hamlen KW, Lin Z (2012) Securing untrusted code via compiler-agnostic binary rewriting. In: Proceedings of the 28th annual computer security applications conference. ACM, pp 299–308
Woeginger GJ (2004) Space and time complexity of exact algorithms: some open problems. In: Rod D, Michael F, Frank D (eds) Parameterized and exact computation. Springer, Berlin, pp 281–290
Wojtczuk R (2001) The advanced return-into-Libc exploits: PaX case study. Phrack Magazine
Xia Y, Liu Y, Chen H, Zang B (2012) CFIMon: detecting violation of control flow integrity using performance counters. In: IEEE/IFIP international conference on dependable systems and networks (DSN 2012). IEEE, pp 1–12
Xiaoyang X, Ghaffarinia M, Wang W, Hamlen KW, Lin Z (2019) CONFIRM: evaluating compatibility and relevance of control-flow integrity protections for modern software. In: 28th USENIX security symposium (USENIX Security 19), Santa Clara, CA, August 2019. USENIX Association, Santa Clara, CA, August, pp 1805–1821
Yee B, Sehr D, Dardyk G, Chen JB, Muth R, Ormandy T, Okasaka S, Narula N, Fullagar N (2009) Native client: a sandbox for portable, untrusted x86 native code. In: 2009 30th IEEE symposium on security and privacy (S&P), pp 79–93
Zhang J, Qi B, Qin Z, Qu G (2019) HCIC: hardware-assisted control-flow integrity checking. IEEE Internet Things J 6(1):458–471
Zhang J, Hou R, Fan J, Liu K, Zhang L, McKee SA (2017) RAGuard: a hardware based mechanism for backward-edge control-flow integrity. In: Proceedings of the computing frontiers conference, CF'17, New York, NY, USA, ACM, pp 27–34
Zhang M, Sekar R (2013) Control flow integrity for COTS binaries. In: Proceeding of the 22nd USENIX security symposium (USENIX security 13), pp 337–352
Zhang C, Wei T, Chen Z, Duan L, Szekeres L, McCamant S, Song D, Zou W (2013) Practical control flow ontegrity and randomization for binary executables. In: 2013 IEEE symposium on security and privacy (S&P). IEEE, pp 559–573
Zhi W, Xuxian J (2010) HyperSafe: a lightweight approach to provide lifetime hypervisor control-flow integrity. In: 2010 IEEE symposium on security and privacy, pp 380–395

## Publisher's Note